Praspel: A Specification Language for Contract-Based Testing in PHP

Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, Abdallah Ben Othman

Praspel: A Specification Language for Contract-Based Testing in PHP

Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti and Abdallah Ben Othman

LIFC / INRIA CASSIS Project – 16 route de Gray - 25030 Besançon cedex, France
Email: {ivan.enderlin, abdallah.ben_othman}@edu.univ-fcomte.fr,
{frederic.dadeau, alain.giorgetti}@univ-fcomte.fr

Abstract. We introduce in this paper a new specification language named Praspel, for PHP Realistic Annotation and SPEcification Language. This language is based on the Design-by-Contract paradigm. Praspel clauses annotate methods of a PHP class in order to both specify their contracts, using pre- and postconditions, and assign realistic domains to the method parameters. A realistic domain describes a set of concrete, and hopefully relevant, values that can be assigned to the data of a program (class attributes and method parameters). Praspel is implemented into a unit test generator for PHP that offers a random test data generator, which computes test data, coupled with a runtime assertion checker, which decides whether a test passes or fails by checking the satisfaction of the contracts at run-time.
Keywords. PHP, Design-by-Contract, annotation language, unit testing, formal specifications.

1 Introduction

Over the years, testing has become the main way to validate software. A challenge is the automation of test generation that aims to unburden the developers from writing their tests manually. Recent development techniques, such as Agile methods, consider tests as first-class citizens, that are written prior to the code. Model-based testing [5] is an efficient paradigm for automating test generation. It considers a model of the system that is used for generating conformance test cases (w.r.t. the model) and computing the oracle (*i.e.* the expected result) that is used to decide whether a test passes or fails.

In order to ease the model description, annotation languages have been designed, firstly introduced by B. Meyer [19], creating the *Design-by-Contract* paradigm. These languages make it possible to express formal properties (invariants, preconditions and postconditions) that directly annotate program entities (class attributes, methods parameters, etc.) in the source code. Many annotation languages exist, such as the Java Modeling Language (JML) [15] for Java, Spec# [3] for C#, or the ANSI-C Specification Language (ACSL) [4] for C. Design-by-Contract considers that a system has to be used in a contractual way: to invoke a method the caller has to fulfil its precondition; in return, the method establishes its postcondition.

Contract-Driven Testing. Annotations can be checked at run time to make sure that the system behaves as specified, and does not break any contract. Contracts are thus well-suited to testing, and especially to unit testing [16]. The idea of Contract-Driven Testing [1] is to rely on contracts for both producing tests, by computing test data satisfying the contract described by the precondition, and for test verdict assessment, by checking that the contract described by the postcondition is ensured after execution. On one hand, method preconditions can be used to generate test data, as they characterize the states and parameters for which the method call is licit. For example, the Jartege tool [20] generates random test data, in accordance with the domain of the inputs of a given Java method, and rejects the values that falsify the precondition. The JML-Testing-Tools toolset [6] uses the JML precondition of a method to identify boundary states from which the Java method under test will be invoked. The JMLUnit [9] approach considers systematic test data for method input parameters, and filters irrelevant ones by removing those falsifying the method precondition. On the other hand, postconditions are employed similarly in all the approaches [6,8,9]. By runtime assertion checking, the postcondition is verified after each method execution, to provide the test verdict.

Contributions. In this paper, we present a new language named Praspel, for *PHP Realistic Annotation and SPEcification Language*. Praspel is a specification language for PHP [21] which illustrates the concept of realistic domains. Praspel introduces Design-by-Contract in PHP, by specifying realistic domains on class attributes and methods. Consequently, Praspel is adapted to test generation: contracts are used for unit test data generation and provide the test oracle by checking the assertions at runtime.

Our second contribution is a test framework supporting this language. This online test generation and execution tool works in three steps: (*i*) the tool generates values for variables according to the contract (possibly using different data generators), (*ii*) it runs the PHP program with the generated values, and (*iii*) the tool checks the contract postcondition to assign the test verdict.

Paper outline. The paper is organized as follows. Section 2 briefly introduces the concept of realistic domains. Section 3 presents an implementation of realistic domains in the Praspel language, a new annotation language for PHP. Section 4 describes the mechanism of automated generation of unit tests from PHP files annotated with Praspel specifications. The implementation of Praspel is described in Section 5. Section 6 compares our approach with related works. Finally, Section 7 concludes and presents our future work.

2 Realistic Domains

When a method precondition is any logical predicate, say from first-order logic, it can be arbitrarily difficult to generate input data satisfying the precondition.

One could argue that the problem does not appear in practice because usual preconditions are only simple logical predicates. But defining what is a "simple" predicate w.r.t. the problem of generating values satisfying it (its *models*) is a difficult question, still partly open. We plan to address this question and to put its positive answers at the disposal of the test community. In order to reach this goal we introduce the concept of realistic domains.

Realistic domains are intended to be used for test generation purposes. They specify the set of values that can be assigned to a data in a given program. Realistic domains are well-suited to PHP, since this language is dynamically typed. Therefore, realistic domains introduce a specification of data types that are mandatory for test data generation. We introduce associated features to realistic domains, and we then present the declination of realistic domains for PHP.

2.1 Features of realistic domains

Realistic domains are structures that come with necessary properties for the validation and generation of data values. Realistic domains can represent all kinds of data; they are intended to specify relevant data domains for a specific context. Realistic domains are more subtle than usual data types (integer, string, array, etc.) and are actually refinement of those latters. For example, if a realistic domain specifies an email address, we can validate and generate strings representing syntactically correct email addresses, as shown in Fig. 1.

Realistic domains display two features, which are now described and illustrated.

Predicability. The first feature of a realistic domain is to carry a characteristic predicate. This predicate makes it possible to check if a value belongs to the possible set of values described by the realistic domain.

Samplability. The second feature of a realistic domain is to propose a value generator, called the *sampler*, that makes it possible to generate values in the realistic domain. The data value generator can be of many kinds: a random generator, a walk in the domain, an incrementation of values, etc.

We now present our implementation of realistic domains in PHP and show some interesting additional principles they obey.

2.2 Realistic domains in PHP

In PHP, we have implemented realistic domains as classes providing at least two methods, corresponding to the two features of realistic domains. The first method is named `predicate($q)` and takes a value `$q` as input: it returns a boolean indicating the membership of the value to the realistic domain. The second method is named `sample()` and generates values that belong to the realistic

```
class EmailAddress extends String {

  public function predicate($q) {
    // regular expression for email addresses
    //  see. RFC 2822, 3.4.1. address specs.
    $regexp = '...';
    if(false === parent::predicate($q))
      return false;

    return preg_match($regexp,$q);
  }

  public function sample() {
    // string of authorized chars
    $chars = 'ABCDEFGHIJKL...';
    // array of possible domain extensions
    $doms  = array('net','org','edu','com');
    $q     = '';

    $nbparts = rand(2, 4);

    for($i = 0; $i < $nbparts; ++$i) {
      if($i > 0)
        // add separator or arobase
        $q .= ($i == $nbparts - 1)
                   ? '@' : '.';

      $len = rand(1,10);
      for($j=0; $j < $len; ++$j) {
        $index = rand(0, strlen($chars) - 1);
        $q .= $chars[$index];
      }
    }
    $q .= '.' .
          $doms[rand(0, count($doms) - 1)];
    return $q;
  }
}
```

Fig. 1. PHP code of the EmailAddress realistic domain

domain. An example of realistic domain implementation in a PHP class is given in Fig. 1. This class represents the `EmailAddress` realistic domain already mentioned. Our implementation of realistic domains in PHP exploit the PHP object programming paradigm and benefit from the following two additional principles.

Hierarchical inheritance. PHP realistic domains can inherit from each other. A realistic domain child inherits the two features of its parent and is able to redefine them. Consequently, all the realistic domains constitute an universe.

Parameterizable realistic domains. Realistic domains may have parameters. They can receive arguments of many kinds. In particular, it is possible to use realistic domains as arguments of realistic domains. This notion is very important for the generation of recursive structures, such as arrays, objects, graphs, automata, etc.

Example 1 (Realistic domains with arguments). The realistic domain `boundinteger(`X`, `Y`)` contains all the integers between X and Y. The realistic domain `string(`L`, `X`, `Y`)` is intended to contain all the strings of length L constituted of characters from X to Y Unicode code-points. In the realistic domain `string(boundinteger(4, 12), 0x20, 0x7E)`, the string length is defined by another realistic domain.

3 PHP Realistic Annotation and Specification Language

Realistic domains are implemented for PHP in Praspel, a dedicated annotation language based on the Design-by-Contract paradigm [19]. In this section, we present the syntax and semantics of the language.

Praspel specifications are written in API documentation comments as shown in Fig. 3, 4, 5 and 6.

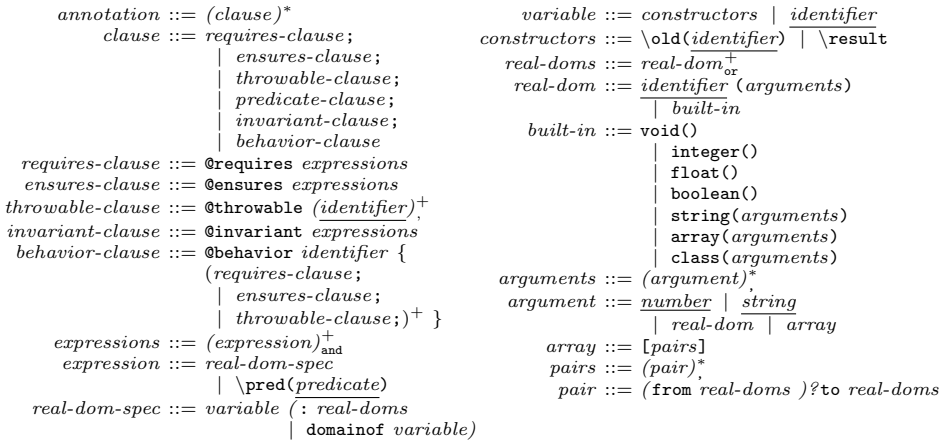

Fig. 2. Grammar of the concrete syntax

Praspel makes it possible to mix informal documentations and formal constraints, called *clauses* and described hereafter. Praspel clauses are ignored by PHP interpreters and integrated development environments. Moreover, since each Praspel clause begins with the standard `@` symbol for API keywords, it is usually well-handled by pretty printers and API documentation generators.

The grammar of Praspel annotations is given in Fig. 2. Notation $(\sigma)?$ means that σ is optional. $(\sigma)^r_s$ represents finite sequences of elements matching σ, in which r is either `+` for one or more matches, or `*` for zero or more matches, and s is the separator for the elements in the sequence.

Underlined words are PHP entities. They are exactly the same as in PHP. A *predicate* is a valid logical PHP expression that returns a boolean. An *identifier* is the name of a PHP class or the name of a method or method parameter. It cannot be the name of a global variable or an attribute, which are respectively prohibited (as bad programming) and defined as invariants in Praspel. The syntax of identifiers strictly follows the syntax of PHP variables.

The other syntactic entities in this grammar are explained in the PHP manual [21]. Praspel expressions are conjunctions of realistic domain assignments and of relations between realistic domains, explained in Section 3.1. The special case of array descriptions is explained in Section 3.2. Finally, clauses are described in Section 3.3.

3.1 Assigning realistic domains to data

We now explain how to declare the realistic domains of method parameters and we give the semantics of these declarations w.r.t. the method inputs and output.

The syntactic construction:

$$\texttt{i:}\ \ t_1(\dots)\ \texttt{or}\ \ \dots\ \ \texttt{or}\ \ t_n(\dots)$$

associates at least one realistic domain (among $t_1(\dots)$, $\dots$, $t_n(\dots)$) to an identifier (here, `i`). We use the expression "domains disjunction" when speaking about syntax, and the expression "domains union" when speaking about semantics. The left-hand side represents the name of some method argument, whereas the right-hand side is a list of realistic domains, separated by the "`or`" keyword.

The semantics of such a declaration is that the realistic domain of the identifier `i` may be $t_1(\dots)$ or $\dots$ or $t_n(\dots)$, *i.e.* it is the union (as in the C language) of the realistic domains $t_1(\dots)$ to $t_n(\dots)$. These realistic domains are (preferably) mutually exclusive.

Example 2 (An identifier with many realistic domains). The declaration:

```
y: integer() or float() or boolean()
```

means that `y` can either be an integer, a floating-point number or a boolean.

The `domainof` operator describes a dependency between the realistic domains of two identifiers. The syntactic construction: *identifier* `domainof` *identifier* creates this relation. The semantics of i `domainof` j is that the realistic domain chosen at runtime for j is the same as for i.

When an object is expected as a parameter, it can be specified using the `class(`C`)` construct, in which C is a string designating the class name.

Example 3 (Use of a class as a realistic domain). The following declaration:

```
o: class('LimitIterator') or class('RegexIterator')
```

specifies that `o` is either an instance of `LimitIterator` or `RegexIterator`.

3.2 Array description

A realistic domain can also be an array description. An array description has the following form:

[from T_1^1(...) or ... or T_i^1(...) to T_{i+1}^1(...) or ... or T_n^1(...),
...
from T_1^k(...) or ... or T_j^k(...) to T_{j+1}^k(...) or ... or T_m^k(...)]

It is a sequence between square brackets "`[`" and "`]`" of pairs separated by symbol "`,`". Each pair is composed of a domain introduced by the keyword "`from`", and a co-domain introduced by the keyword "`to`". Each domain and co-domain is a disjunction of realistic domains separated by the keyword "`or`". The domain is optional.

The semantics of an array description depends of the realistic domain where it is used. We detail this semantics in the most significant case, when the array description is a parameter of the realistic domain `array`. Notice that an array description and the realistic domain `array` are different. The realistic domain `array` has two arguments: the array description and the array size.

Example 4 (Array specification). Consider the following declarations:

```
a1: array([from integer() to boolean()], boundinteger(7, 42))
a2: array([to boolean(), to float()], 7)
a3: array([from integer() to boolean() or float()], 7)
a4: array([from string(11) to boolean(), to float() or integer()], 7)
```

`a1` describes an homogeneous array of integers to booleans. The size of the yielded array is an integer between 7 and 42. In order to produce a fixed-size array, one must use an explicit constant, as in the subsequent examples. `a2` describes two homogeneous arrays: either an array of booleans, or an array of floats, but not both. In all cases, the yielded array will have a size of 7. If no realistic domain is given as domain (*i.e.* if the keyword "`from`" is not present), then an auto-incremented integer (an integer that is incremented at each sampling) will be yielded. `a2` is strictly equivalent to `array([to boolean()], 7) or array([to float()], 7)`. `a3` describes an heterogeneous array of booleans and floats altogether. Finally, `a4` describes either an homogeneous array of strings to booleans or an heterogeneous array of floats and integers.

3.3 Designing Contracts in Praspel

This section describes the syntax and semantics of the part of Praspel that defines contracts for PHP classes and methods. Basically, a contract clause is either the assignment of a realistic domain to a given data, or it is a PHP predicate that provides additional constraints over variables values (denoted by the `\pred` construct).

Invariants. In the object programming paradigm, class attributes may be constrained by properties called "invariants". These properties must hold before and after any method call and have the following syntax.

`@invariant` I_1 `and ... and` I_p`;`

Classically, invariants have to be satisfied after the creation of the object, and preserved through each method execution (*i.e.* assuming the invariant holds before the method, then it has to hold once the method has terminated). Invariants are also used to provide a realistic domain to the attributes of an object.

```
class C {
  /**
   * @invariant a: boolean();
   */
  protected $a;
}
```

Fig. 3. Example of invariant clause

Example 5 (Simple invariant). The invariant in Fig. 3 specifies that the attribute `a` is a boolean before and after any method call.

Method contracts. Praspel makes it possible to express contracts on the methods in a class. The contract specifies a precondition that has to be fulfilled for the method to be executed. In return, the method has to establish the specified postcondition. The contract also specifies a set of possible exceptions that can be raised by the method. The syntax of a basic method contract is given in Fig. 4. The semantics of this contract is as follows.

```
/**
 * @requires  R1 and ... and Rn;
 * @ensures   E1 and ... and Em;
 * @throwable T1, ..., Tp;
 */
function foo ( ... ) { ... }
```

Fig. 4. Syntax of Method Contracts

Each R_x $(1 \leq x \leq n)$ represents either the assignment of a realistic domain to a data, or a PHP predicate that provides a precondition. The caller of the method must guarantee that the method is called in a state where the properties conjunction $R_1 \wedge \ldots \wedge R_n$ holds (meaning that the PHP predicate is true, and the value of the data should match their assigned realistic domains). By default, if no `@requires` clause is declared, the parameter is implicitly declared with the `undefined` realistic domain (which has an always-true predicate and sample an integer by default).

Each E_z $(1 \leq z \leq m)$ is either the assignment of a realistic domain to a data (possibly the result of the method) after the execution of the method, or an assertion specified using a PHP predicate that specifies the postcondition of the method. The method, if it terminates normally, should return a value such that the conjunction $E_1 \wedge \ldots \wedge E_m$ holds.

Each T_y $(1 \leq y \leq p)$ is an exception name. The method may throw one of the specified exceptions $T_1 \vee \ldots \vee T_p$ or a child of these ones. By default, the specification does not authorize any exception to be thrown by the method.

```
/**
 * @requires  x: boundinteger(0,42);
 * @ensures   \result: eveninteger()
 *         and \pred(x >= \old(x));
 * @throwable FooBarException;
 */
function foo ( $x ) {
  if($x === 42)
    throw new FooBarException();
  return $x * 2;
}
```

Fig. 5. Example of Simple Contract

Postconditions (`@ensures` clauses) usually have to refer to the method result and to the variables in the pre-state (before calling the function). Thus, PHP expressions are extended with two new constructs: `\old(`e`)` denotes the value of expression e in the pre-state of the function, and `\result` denotes the value returned by the function. Notice that our language does not include first-order logic operators such as universal or existential quantifiers. Nevertheless, one can easily simulate such operators using a dedicated boolean function, that would be called in the PHP predicate.

```
/**
 * @requires R_1 and ... and R_n;
 * @behavior α {
 *   @requires  A_1 and ... and A_k;
 *   @ensures   E_1 and ... and E_j;
 *   @throwable T_1, ..., T_t;
 * }
 * @ensures   E_{j+1} and ... and E_m;
 * @throwable T_{t+1}, ..., T_l;
 */
function foo ( $x_1... ) { body }
```

(a)

```
/**
 * @requires x: integer();
 * @behavior foo {
 *   @requires  y: positiveinteger()
 *          and  z: boolean();
 * }
 * @behavior bar {
 *   @requires  y: negativeinteger()
 *          and  z: float();
 *   @throwable BarException;
 * }
 * @ensures \result: boolean();
 */
function foo ( $x, $y, $z ) { ... }
```

(b)

Fig. 6. Behavioral Contracts

Example 6 (Simple contract). Consider the example provided in Fig. 5. This function `foo` doubles the value of its parameter `$x` and returns it. In a special case, the function throws an exception.

Behavorial clauses. In addition, Praspel makes it possible to describe explicit *behaviors* inside contracts.

A behavior is defined by a name and local `@requires`, `@ensures`, and `@throwable` clauses (see Fig. 6(a)). The semantics of behavioral contracts is as follows.

The caller of the method must guarantee that the call is performed in a state where the property $R_1 \wedge \ldots \wedge R_n$ holds. Nevertheless, property $A_1 \wedge \ldots \wedge A_k$ should also hold. The called method establishes a state where the property $(A_1 \wedge \ldots \wedge A_k \Rightarrow E_1 \wedge \ldots \wedge E_j) \wedge E_{j+1} \wedge \ldots \wedge E_m$ holds, meaning that the postcondition of the specified behavior only has to hold if the precondition of the behavior is satisfied. Exceptions T_i $(1 \leq i \leq t)$ can only be thrown if the preconditions $R_1 \wedge \ldots \wedge R_n$ and $A_1 \wedge \ldots \wedge A_k$ hold.

The `@behavior` clause only contains `@requires`, `@ensures` and `@throwable` clauses. If a clause is declared or used outside a behavior, it will automatically be set into a default/global behavior. If a clause is missing in a behavior, the clause in the default behavior will be chosen.

Example 7 (Behavior with default clauses). The specification in Fig. 6(b) is an example of a complete behavioral clause. This contract means: the first argument `$x` is always an integer and the result is always a boolean, but if the second argument `$y` is a positive integer, then the third argument `$z` is a boolean (behavior `foo`), else if `$y` is a negative integer, and the `$z` is a float and then the method may throw an exception (behavior `bar`).

4 Automated Unit Test Generator

The unit test generator works with the two features provided by the realistic domains. First, the sampler is implemented as a random data generator, that

```
public function foo ($x1 ...) {

  $this->foo_pre(...);

  // evaluation of \old(e)

  try {

   $result = $this->foo_body($x1 ... );

  }
 catch ( Exception $exc ) {
   $this->foo_exception($exc);
   throw $exc;
 }

 $this->foo_post($result, ...);

 return $result;
}
```

```
public function foo_pre(...) {

  return    verifyInvariants(...)
         && verifyPreCondition(...);
}

public function foo_post(...) {

  return    verifyPostCondition(...)
         && verifyInvariants(...);
}

public function foo_exception($e) {

  return    verifyException($e)
         && verifyInvariants(...);
}

public function foo_body($x1 ... ) ...
```

Fig. 7. PHP Instrumentation for Runtime Assertion Checking

satisfies the precondition of the method. Second, the predicate makes it possible to check the postcondition (possibly specifying realistic domains too) at runtime after the execution of the method.

4.1 Test Verdict Assignment using Runtime Assertion Checking

The test verdict assignment is based on the runtime assertion checking of the contracts specified in the source code. When the verification of an assertion fails, a specific error is logged. The runtime assertion checking errors (a.k.a. Praspel failures) can be of five kinds. (*i*) precondition failure, when a precondition is not satisfied at the invocation of a method, (*ii*) postcondition failure, when a postcondition is not satisfied at the end of the execution of the method, (*iii*) throwable failure, when the method execution throws an unexpected exception, (*iv*) invariant failure, when the class invariant is broken, or (*v*) internal precondition failure, which corresponds to the propagation of the precondition failure at the upper level.

The runtime assertion checking is performed by instrumenting the initial PHP code with additional code which checks the contract clauses. The result of the code instrumentation of a given method `foo` is shown in Fig. 7. It corresponds to the treatment of a behavioral contract, as shown in Fig. 6. The original method `foo` is duplicated, renamed (as `foo_body`) and substituted by a new `foo` method which goes through the following steps:

- First, the method checks that the precondition is satisfied at its beginning, using the auxiliary method `foo_pre`.
- Second, the `\old` expressions appearing in the postconditions are evaluated and stored for being used later.

– Third, the replication of the original method body is called. Notice that this invocation is surrounded by a try-catch block that is in charge of catching the exception that may be thrown in the original method.
– Fourth, when the method terminates with an exception, this exception is checked against the expected ones, using the auxiliary method `foo_exception`. Then the exception is propagated so as to preserve the original behavior of the method.
– Fifth, when the method terminates normally, the postconditions and the invariants are checked, using the auxiliary method `foo_post`.
– Sixth, and finally, the method returns the value resulting of the execution of `foo_body`.

Test cases are generated and executed online: the random test generator produces test data, and the instrumented version of the initial PHP file checks the conformance of the code w.r.t. specifications for the given inputs. The test succeeds if no Praspel failure (listed previously) is detected. Otherwise, it fails, and the log indicates where the failure has been detected.

4.2 Random Test Data Generation

To generate test data, we rely on a randomizer, a sampling method that is in charge of generating a random value for a given realistic domain. The randomizer works with the realistic domain assignments provided in the `@requires` clauses of the contracts.

Assume that the precondition of a method specifies the realistic domain of parameter `i` as follows: `@requires i:` $t_1(\ldots)$ `or ... or` $t_n(\ldots)$`;`. When this method is randomly tested, a random value for `i` is generated in the domain of one of its n declared realistic domains. If $n \geq 2$, then a realistic domain $t_c(\ldots)$ is first selected among $t_1(\ldots)$, $\ldots$, $t_n(\ldots)$ by uniform random generation of c between 1 and n. Then, the randomizer generates a value of domain $t_c(\ldots)$ for i using the sampling method provided by this realistic domain.

When the data to be generated is an object of class C, the invariant of class C is analyzed in order to recover the realistic domains associated to the class attributes, and it recursively generates a data value for each class attribute. An instance of class C is created and the generated data values are assigned to the class attributes.

By default, the test case generation works by rejection of irrelevant values, as described in the simplified algorithm in Fig. 8. This algorithm has three parameters. The first one, *nbTests*, represents the number of tests the user wants to generate. The second parameter, *maxTries*, is introduced in order to ensure that the generator stops if all the yielded values are rejected. Indeed, the generator may fail to yield a valid value because the preconditions lay down too strong constraints. The third and last parameter, *methods*, represents the set of methods to test.

For many realistic domains this default test generation method can be overloaded by more efficient methods. We have already implemented the following enhancements:

```
function generateTests(nbTests, maxTries, methods)
begin
  tests ← 0
  do
    tries ← 0
    f ← select method under test amongst methods
    do
      tries ← tries + 1
      for each parameter p of f do
        t ← select realistic domain of p
        i ← generate random value ∈ [1..card(t)]
        v ← i^th value of realistic domain t
        assign value v to parameter p
      done
    while f precondition fails ∧ tries ≤ maxTries
    if tries ≤ maxTries then
      run test case / keep test case
      tests ← tests + 1
    end if
  while tests ≤ nbTests
end
```

Fig. 8. Test Cases Generation Algorithm

- Generation by rejection is more efficient when the rejection probability is low. Therefore, the hierarchy of realistic domains presented in Section 2.2 is constructed so that each class restricts the domain of its superclass as little as possible, and rejection always operates w.r.t. this direct superclass for more efficiency.
- Realistic domains representing intervals are sampled without rejection by a direct call to a uniform random generator, in fact two generators: a discrete and a continuous one.

The users are free to add their own enhancements and we plan to work in this direction in a near future.

5 Tool Support and Experimentation

The Praspel tool implements the principles described in this paper. Praspel is freely available at `http://hoa-project.net/`. It is developed in Hoa [12] (since release 0.5.5b), a framework and a collection of libraries developed in PHP. Hoa combines a modular library system, a unified streams system, generic solutions, etc. Hoa also aims at bridging the gap between research and industry, and tends to propose new solutions from academic research, as simple as possible, to all users. Praspel is a native library of Hoa (by opposition to user libraries). This deep integration ensures the durability and maintenance of Praspel. It is provided with an API documentation, a reference manual, a dedicated forum, etc. Moreover, we get feedbacks about the present research activity through Hoa user community.

A demonstration video of the Praspel tool is also available at the following address: `http://hoa-project.net/Video/Praspel.html`. As shown in this demonstration, the Praspel language is easy to learn and to use for developers. The

tool support makes it possible to produce test data and run test cases in a dedicated framework with little effort. Praspel has been presented and demonstrated at the ForumPHP'10 conference, which gathers the PHP community, and was received very favourably.

We applied Praspel to web application validation. In such applications, PHP functions are used to produce pieces of HTML code from simple inputs. Being also teachers, we have decided to test the (buggy!) code of our students of the "Web Languages" course. Using a dedicated library of Hoa to build LL(k) parsers, we easily designed the realistic domains associated with the expected output HTML code, that had to be well-structured, and respecting some structural constraints (e.g. exactly one `<option>` of a `<select>` has to be set by default). We analysed their functions using this mechanism.

To save space, we have reported this experimentation on a web page at the following address: `http://lifc.univ-fcomte.fr/home/~fdadeau/praspel.html`, along with the links to download the Hoa framework (implementing Praspel), and some samples of the possibilities of Praspel.

6 Related Works

Various works consider Design-by-Contract for unit test generation [6,9,10,14,17]. Our approach is inspired by the numerous works on JML [15]. Especially, our test verdict assignment process relies on Runtime Assertion Checking, which is also considered in JMLUnit [9], although the semantics on exceptions handling differs. Recently, JSConTest [14] uses Contract-Driven Testing for JavaScript. We share a common idea of adding types to weakly typed scripting languages (JavaScript vs PHP). Nevertheless our approach differs, by considering flexible contracts, with type inheritance, whereas JSConTest considers basic typing informations on the function profile and additional functions that must be user-defined. As a consequence, due to a more expressive specification language, Praspel performs more general runtime assertion checks. ARTOO [10] (Adaptative Random Testing for Object-Oriented software) uses a similar approach based on random generation involving contracts written in Eiffel [18], using the AutoTest tool. ARTOO defines the notion of distance between existing objects to select relevant input test data among the existing objects created during the execution of a program. Our approach differs in the sense that object parameters are created on-the-fly using the class invariant to determine relevant attributes values, in addition to the function precondition. Praspel presents some similarities with Eiffel's types, especially regarding inheritance between realistic domains. Nevertheless, realistic domains display the two properties of predicability and samplability that do not exist in Eiffel. Moreover, Praspel adds clauses that Eiffel contracts do not support, as `@throwable` and `@behavior`, which are inspired from JML.

Our test generation process, based on random testing, is similar to Jartege [20] for JML. Jartege is able to generate a given number of tests sequences of a given

length, for Java programs with JML specifications. Jartege uses the type of input parameters to compute random values for method parameters, that are then checked against the precondition of the method, using an online test generation approach.

Also for JML, Korat [7] uses a user-defined boolean Java function that defines a valid data structure to be used as input for unit testing. A constraint solving approach is then used to generate data values satisfying the constraints given by this function, without producing isomorphic data structures (such as trees). Our approach also uses a similar way to define acceptable data (the predicate feature of the realistic domains). Contrary to Korat, which automates the test data generation, our approach also requires the user to provide a dedicated function that generates data. Nevertheless, our realistic domains are reusable, and Praspel provides a set of basic realistic domains that can be used for designing other realistic domains. Java PathFinder [23] uses a model-checking approach to build complex data structures using method invocations. Although this technique can be viewed as an automation of our realistic domain samplers, its application implies an exhausive exploration of a system state space. Recently, the UDITA language [13] makes it possible to combine the last two approaches, by providing a test generation language and a method to generate complex test data efficiently. UDITA is an extension of Java, including non-deterministic choices and assumptions, and the possibility for the users to control the patterns employed in the generated structures. UDITA combines generator- and filter-based approaches (respectively similar to the sampler and characteristic predicate of a realistic domain).

Finally, in the domain of web application testing, the Apollo [2] tool makes it possible to generate test data for PHP applications by code analysis. The tests mainly aim at detecting malformed HTML code, checked by an common HTML validator. Our approach goes further as illustrated by the experimentation, as it makes it possible not only to validate a piece of HTML code (produced by a Praspel-annotated function/method), but also to express and check structural constraints on the resulting HTML code. On the other hand, the test data generation technique proposed by Apollo is of interest and we are now investigating similar techniques in our test data generators.

7 Conclusion and Future Works

In this paper, we have introduced the concept of realistic domain in order to specify test data for program variables. Realistic domains are not types but may replace them in weakly-typed languages, such as Web languages like PHP, our implementation choice. Realistic domains are a (realistic) trade-off between two extreme ways to specify preconditions: a too coarse one with types and a too fine one with first-order formulas. They provide two useful features for automated test generation: *predicability* and *samplability*. Predicability is the ability to check the realistic domain of a data at run time, whereas samplability provides a means to automatically generate data values matching a given real-

istic domain. Realistic domains are specified as annotations in the code of the language. We have implemented these principles on PHP, in a dedicated specification language named Praspel, based on the Design-by-Contract paradigm. This approach is implemented into a test generator that is able to (*i*) generate automatically unit test data for PHP methods, using a random data generator that produces input parameter values that satisfy the precondition, (*ii*) run the tests and (*iii*) check the postcondition to assign the test verdict. Praspel implements the two features of runtime assertion checking and test data generation independently. As a consequence, the user is not restricted to use a random test data generator and can write his own test data generator.

We are currently investigating the improvement of our data generator, to replace randomly generated values by a constraint-based approach [11] for which the realistic domains of input parameters would be used to define data domains. Constraint solving techniques will then be employed to produce test data that satisfy the precondition of the method, as in Pex [22]. We are also planning to consider a grey-box testing approach that would combine a structural analysis of the code of PHP methods, coupled with the definition of the realistic domains of their input parameters. Finally, we are interested in extending (and generalizing) the concept of realistic domains to other programming languages, especially those already providing a type system, to illustrate the benefits of this concept.

References

1. B. K. Aichernig. Contract-based testing. In *Formal Methods at the Crossroads: From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 34–48. Springer, 2003.
2. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M.D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 261–272, New York, NY, USA, 2008. ACM.
3. M. Barnett, K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Proceedings of the International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04)*, volume 3362 of *LNCS*, pages 49–69, Marseille, France, March 2004. Springer-Verlag.
4. P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, 2008.
5. B. Beizer. *Black-box testing: techniques for functional testing of software and systems.* John Wiley & Sons, Inc., New York, NY, USA, 1995.
6. F. Bouquet, F. Dadeau, and B. Legeard. Automated Boundary Test Generation from JML Specifications. In *FM'06, 14th Int. Conf. on Formal Methods*, volume 4085 of *LNCS*, pages 428–443, Hamilton, Canada, August 2006. Springer.
7. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing based on Java Predicates. In *ISSTA'02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM.
8. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In

J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *International Symposium of Formal Methods (FM'2005)*, volume 3582 of *LNCS*, pages 542–547. Springer, 2005.
9. Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference*, volume 2374 of *LNCS*, pages 231–255, Berlin, June 2002. Springer.
10. I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: Adaptive Random Testing for Object-Oriented Software. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 71–80, New York, NY, USA, 2008. ACM.
11. R. A. DeMillo and A. J. Offutt. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.
12. I. Enderlin. Hoa Framework project, 2010. URL: http://hoa-project.net.
13. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 225–234, New York, NY, USA, 2010. ACM.
14. P. Heidegger and P. Thiemann. Contract-Driven Testing of JavaScript Code. In *TOOLS 2010 - 48th Int. Conf. on Objects, Models, Components, Patterns*, volume 6141 of *LNCS*, pages 154–172, 2010.
15. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
16. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In F. S. de Boer, Bonsangue M. M., S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Lieden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *LNCS*, pages 262–284. Springer, Berlin, 2003.
17. P. Madsen. Unit Testing using Design by Contract and Equivalence Partitions. In *XP'03: Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering*, pages 425–426, Berlin, Heidelberg, 2003. Springer.
18. B. Meyer. Eiffel: programming for reusability and extendibility. *SIGPLAN Not.*, 22(2):85–94, 1987.
19. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
20. C. Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In R. Reussner, J. Mayer, J.A. Stafford, S. Overhage, S. Becker, and P.J. Schroeder, editors, *First Int. Conf. on the Quality of Software Architectures, QoSA 2005 and Second Int. Workshop on Software Quality, SOQUA 2005*, volume 3712 of *LNCS*, pages 242–256, Erfurt, Germany, September 2005. Springer.
21. PHP Group. The PHP website, 2010. URL: http://www.php.net.
22. N. Tillmann and J. de Halleux. Pex: White box test generation for .net. In B. Beckert and R. Hhnle, editors, *Tests and Proofs*, volume 4966 of *LNCS*, pages 134–153. Springer Berlin / Heidelberg, 2008.
23. W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, 2004.